Taylor & Francis
Taylor & Francis Group

Cybernetics and Systems
An International Journal

ISSN: 0196-9722 (Print) 1087-6553 (Online) Journal homepage: http://www.tandfonline.com/loi/ucbs20

Solution Modeling Using Postfix Genetic Programming

Vipul K. Dabhi & Sanjay Chaudhary

To cite this article: Vipul K. Dabhi & Sanjay Chaudhary (2015) Solution Modeling Using Postfix Genetic Programming, Cybernetics and Systems, 46:8, 605-640, DOI: 10.1080/01969722.2015.1058662

To link to this article: http://dx.doi.org/10.1080/01969722.2015.1058662

Cybernetics and Systems: An International Journal, 46:605–640

ISSN: 0196-9722 print/1087-6553 online
DOI: 10.1080/01969722.2015.1058662

Solution Modeling Using Postfix Genetic Programming

VIPUL K. DABHI[1] and SANJAY CHAUDHARY[2]

[1]*Department of Information Technology, Dharmsinh Desai University, Nadiad, Gujarat, India*

[2]*Institute of Engineering and Technology, Ahmedabad University, Ahmedabad, Gujarat, India*

This article introduces Postfix Genetic Programming (GP), a postfix notation-based GP, approach to symbolic regression for solving empirical modeling problems. The main features of Postfix-GP are presented. These features include (1) postfix-based, variable-length individual-representation and its stack-based evaluation and (2) subtree crossover operator and semantic-aware subtree crossover operator. The article also presents two different constant creation approaches to evolve useful numeric constants for symbolic regression problems. The first approach uses an explicit list of constants and the second allows the Postfix-GP algorithm to evolve constants. Use of Postfix-GP as a solution modeling tool is demonstrated by solving two function identification problems and two deterministic chaotic time series modeling problems. Effectiveness of the evolved models is tested by statistically analyzing their performance on training and out-of-sample datasets of the problems. Experimental results on test problems suggest that Postfix-GP offers a new possibility for solving empirical modeling problems. We also compared the performance of Postfix-GP with the lil-GP system for all four problems. The results suggest that the quality of solutions found by Postfix-GP was better than that found by lil-GP, for the tested problems.

KEYWORDS empirical modeling, genetic programming, postfix genetic programming, semantic-aware subtree crossover, symbolic regression

Address correspondence to Vipul K. Dabhi, Department of Information Technology, Dharmsinh Desai University, Nadiad-387001, Gujarat, India. E-mail: vipul.k.dabhi@gmail.com

Color versions of one or more of the figures in the article can be found online at www.tandfonline.com/ucbs.

INTRODUCTION

Many scientific applications require that a multivariate experimental dataset be converted into a model (mathematical equation) that can explain hidden relationships between a dependent variable and independent variables. Developing a mathematical model of a process from experimental data is known as an *empirical modeling*. The developed model can provide better understanding of the behavior of the underlying process that produced the experimental data.

Traditional mathematical techniques assume a model's structure (i.e., degree of polynomials) first, which demands good theoretical (domain) knowledge, and then they solve the coefficients of the model by the least square method. It is difficult to solve empirical modeling problems using traditional mathematical techniques because it is difficult to determine the good structure of a model manually and a priori. Moreover, traditional mathematical techniques are unsuitable for solving empirical modeling problems because of their nonlinearity and multimodality properties. It should be noted that one has to fix the structure of the nonlinear models as well while using these models for solving the empirical modeling problems. For example, if a neural network is selected for modeling the data, one has to fix the number of layers, activation functions, and number of neurons in each layer. Finding both the structure and the appropriate numeric coefficients of a model simultaneously is a real challenge. There are two approaches for dealing with this challenge: (i) generate all possible models and select the best out of them (ii) transform the problem into an optimization problem and use heuristics to search intelligently for the optimal model. The problem with the first approach is that there is a huge set of models that can fit a given finite dataset. This makes model search space very large, and it is not possible to evaluate all models in order to select the best from the set. Therefore, the second approach is preferred over the first. However, use of the second approach demands an artificial expert that can use heuristics to generate an optimal model for the given experimental data by intelligently searching the space of all possible models.

This study suggests use of a symbolic optimization technique called symbolic regression to develop the optimal models (solutions) from the experimental data. The symbolic regression technique determines the best model from a large class of candidates, in which candidates are explicit symbolic formulas. The technique does not decide the structure of a model in advance, but it finds the appropriate structure and numeric coefficients of the model through a search algorithm. We have developed a Postfix-GP framework, a postfix-based genetic programming system, to perform symbolic regression. The framework builds different models using building blocks provided by the user and seeks for the optimal model from a large set of models, where models are explicit symbolic equations. Standard optimization techniques (i.e., least square regression, gradient descent

method, neural network) cannot be useful to search the optimal model because of highly nonlinear and discontinuous search space of model structures (Babovic and Keijzer 2000).

Evolutionary algorithms (EAs) refer to the group of computational techniques, that apply the theory of natural selection to a population of individuals in order to produce better individuals. Yao and Xu mentioned the following advantages of EA over conventional optimization techniques: "conceptual and computational simplicity, broad applicability, parallelism, robustness to dynamic environments, and ability to solve problems with no known solutions" (Yao and Xu 2006). In EAs, *genotype* refers to the structure and content of an individual, whereas a *phenotype* models the behavior of an individual's genotype. Thus, the phenotype of an individual depends on its genotype. Two important classes of EAs are (i) genetic algorithm (GA) and (ii) genetic programming (GP). The standard GA (Holland 1992) represents an individual using a fixed length binary string. The individual representation approach of GA is not suitable for solving empirical modeling problems, because it does not permit the model structure to change during evolution. Standard GP (Koza 1992) employs a variable length, tree structure scheme for an individual representation. The representation scheme of GP is more general and flexible than GA, because it makes fewer assumptions about the structure of an individual and permits the structure to vary during evolution. Therefore, the GP approach is selected to develop nonlinear models from the given input–output dataset. The individual representation scheme of standard GP does not allow multiple genes per individual in that each gene can represent a small subexpression (partial solution) to build a hierarchical solution. Gene expression programming (GEP) (Ferreira 2001), is a GP variant, in which an individual is usually composed of more than one gene and they are of equal length. GEP genes are composed of two different domains: a head and a tail. The head incorporates elements that represent both functions and terminals, whereas the tail consists of only terminals. Figure 1 shows a GEP individual. Genetic operators such as recombination, mutation, and transposition are applied to the linear structure

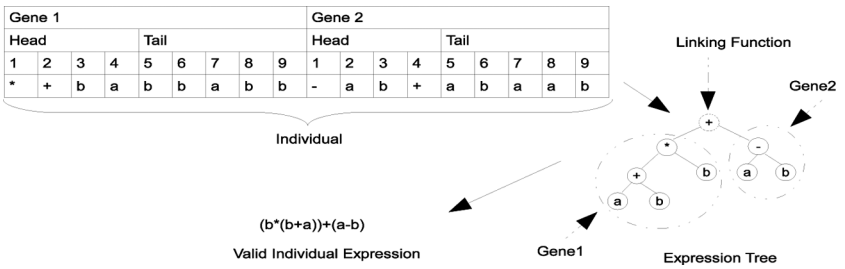

FIGURE 1 Gene expression programming (GEP) individual representation.

(genotype) of an individual, which will be translated later into an expression tree (phenotype). The fundamental difference between GA, GP, and GEP resides in the individual representation schemes used by these approaches.

In this article, we introduce and use our earlier proposed Postfix-GP framework (Dabhi and Vij 2011) for solving empirical modeling problems. Postfix-GP adopts postfix notation for an individual representation and evaluation. The genotype–phenotype mapping mechanism follows the convention of postfix notation. Evaluation of individuals in Postfix-GP does not require construction and traversal of an expression tree (as compared to traditional GP). Postfix-GP uses a stack to decode and evaluate individuals. Representation of individuals in postfix (linear) notation and use of a stack are beneficial for reducing time-space complexity of the GP algorithm. The article uses the terms *individual*, *solution*, and *model* interchangeably.

This study is an extension of our previous work (Dabhi and Vij 2011; Dabhi and Chaudhary 2013), in which we proposed a new individual representation approach (postfix) and algorithm for semantic-aware subtree crossover. We reported initial results in our previous work (Dabhi and Vij 2011; Dabhi and Chaudhary 2013). This article makes the following main contributions: (1) It presents analysis of design choices for the Postfix-GP framework and thoroughly discusses the internals of the Postfix-GP framework compared to previous work. (2) We present algorithms for (i) calculating the end-position of a subtree, (ii) generating phenotypic diverse initial population, and (iii) modified implementation of semantic-aware subtree crossover. (3) The test problems considered in this work are different than those presented in our previous work and are more complex to solve. Moreover, additional statistical measures (correlation coefficient and normalized mean square error) are used to measure the performance (quality) of the evolved solutions. (4) One-step-ahead and multistep-ahead prediction using evolved solutions are performed to measure the performance of the evolved solutions on an out-of-sample dataset. This step was not performed in our earlier work. (5) A performance comparison of Postfix-GP with a conventional GP system, lil-GP, is presented.

The rest of this article is organized as follows: "Design Issues for Building a GP System" presents design issues, alternatives used by GP practitioners to tackle these issues, and a selection of design alternatives used by Postfix-GP. "Postfix Genetic Programming Framework" explains Postfix-GP with its genetic operators, in detail. Experiment setup and results are presented next. "Comparison of Results Obtained by Postx-GP and lil-GP" compares the results of Postfix-GP with those obtained using lil-GP. "Conclusions" are presented in the final section.

DESIGN ISSUES FOR BUILDING A GP SYSTEM

The following design issues need careful attention when designing a GP system: (i) efficiency of GP system, (ii) prevention of premature convergence

of GP, (iii) prevention of evolution of overfitted and bloated solutions, and (iv) efficient constant creation methods. These design issues are discussed in more detail in Dabhi and Chaudhary (2015), O'Neill et al. (2010).

Alternatives for Improving Efficiency of a GP System

One of the issues with GP is that it takes large computation time for evolving solutions (models). The majority of computational time is spent on fitness evaluation of solutions generated over a number of generations. The following three approaches are used by the GP community to reduce this computation time and to improve efficiency of GP algorithm: (i) partial fitness evaluation, (ii) better individual representation, and (iii) subtree caching.

The partial fitness evaluation approach uses the subset of the training dataset rather than the whole training dataset to evaluate the fitness of every individual over a number of generations. Zhang and Cho (1999) applied this approach on symbolic regression problems. They increased the training subset size as evolution progressed for fitness calculation. Smits and Vladislavleva (2006) applied the concept of ordinal optimization in Pareto-GP for partial fitness evaluation of individuals in order to reduce computational effort.

Selection of data structures plays an important role in the implementation of GP, because the selected data structure can have an impact on the performance (search efficiency) of GP (Altwaijry and Menai 2012). For tree-based individual representation, the node evaluation function is called recursively to calculate the fitness of an individual. The evaluation function involves following operations that increase the computational overhead (Tokui and Iha 1999): (i) traversal of nodes using pointers and (ii) parsing of node types. Moreover, each node requires additional memory to store the pointer. Instead of a conventional pointer (tree)-based representation, Tokui and Iha (1999) proposed a linear string-based (in prefix ordering) individual representation to improve the performance of the GP algorithm. The linear representation reduces both fitness evaluation time and memory requirement of individuals. Handley (1994) proposed a subtree caching mechanism to reduce the fitness evaluation time of the individual by caching fitness values of selected subtrees. The cached fitness values of selected subtrees of parents are reused while evaluating the offspring. Keijzer (2004) introduced top-down and bottom-up schemes for the subtree caching. The top-down scheme promotes caching of large subtrees, whereas the bottom-up approach promotes caching of small subtrees.

Alternatives for Preventing Premature Convergence

Premature convergence happens when the individuals of the population get trapped into a local optima and no improvement in fitness is noticed over

consecutive generations. Premature convergence hinders the GP at the generation of the optimal solution for the problem. The following approaches are used by the GP community to prevent premature convergence of the GP algorithm: (i) designing an intelligent crossover operator to improve the diversity of the population and (ii) archiving the best solutions, found so far.

One of the solutions to the problem of premature convergence is to improve the search in the solution space by preserving diversity among individuals of the population. By improving the diversity, the population can cover a large part of the solution space. The population diversity is defined as a percentage of structurally or behaviorally distinct individuals within a population at a given generation (Rosca 1995). GP researchers have used different intelligent crossover operators to improve the population diversity. Gustafson, Burke, and Krasnogor (2005) proposed a "no same-mate selection" mechanism to improve the population diversity. This mechanism avoids mating between two individuals with the same fitness values. The mechanism is helpful in reducing "no changes to fitness" events, often occurring after a crossover operation.

Uy et al. (2009) used the semantic information of two subtrees, to be swapped, to improve the search property of the crossover operator. The semantic of a real-valued mathematical equation is the result it generates with respect to a set of input values. The semantic-aware crossover (SAC) checks the semantic equivalence of two subtrees before performing the crossover operation. The SAC operator improves the population diversity by avoiding swapping of semantically equivalent subtrees. Semantic similarity-based crossover (SSC) (Uy et al. 2011) is an extension of SAC. The technique presumes that exchange of subtrees is good if the subtrees are semantically different, but not too much different (Uy et al. 2011). SSC places bound to control the semantic distance between subtrees to be swapped by the crossover (Uy et al. 2011).

Laumanns et al. (2002) used a finite-sized archive to preserve "best-so-far" individuals. The archive gets updated at every generation. The archive is useful in exploiting good individuals over successive generations.

Alternatives for Prevention of Evolution of Bloated Solutions

The phenomenon of increase in average size of an individual without a corresponding increase in fitness is known as *bloat*. An evolved individual with positive bloat indicates that it is larger than it needs to be. Bloat has a negative effect on the performance of GP, because large individuals are computationally expensive to further evolve and are difficult to comprehend. The approaches used by GP researchers to overcome the problem of bloat include (i) code editing, (ii) size and depth limits, and (iii) an antibloat selection mechanism.

The code editing approach (Koza 1992) simplifies the individual by removing redundant code. Code editing can be performed at every generation. The size/depth limit approach checks the size/depth of generated offspring (Silva and Costa 2009). If the offspring exceeds the size/depth limit, then it gets discarded and the best of the parents is returned by the genetic operator. Parsimony pressure and multiobjective selection approaches are used by the GP practitioners to avoid the evolution of bloated individuals. The parsimony pressure approach penalizes the fitness of an individual based on its size. Parsimony pressure selection combines two objectives, size and fitness of an individual, into a single objective, whereas multiobjective selection keeps these objectives separate.

Alternatives for Efficient Constant Creation

Finding useful numerical constants is an open issue in GP. It is hypothesized that the terminal set is defined over a constant range so that it contains a set of useful constants needed to represent the solution. Two approaches for constant creation (Ferreira 2003) are (i) creating constants from scratch, and (ii) providing an explicit list of constants that can be part of an individual.

The first approach does not use an explicit list of constants, rather it lets the GP algorithm find the constants. Ferreira (2003) applied this approach for solving three symbolic regression problems. The second approach requires an explicit list or range of numerical constants as part of the terminal set. Ryan and Keijzer (2003) proposed two constant mutation approaches: creep mutation and random mutation. Creep (or local) mutation changes the values of constants in a stepwise fashion, whereas random mutation selects a new random value uniformly from the defined range. A hybrid system that combines GA and GP for solving symbolic regression problems is proposed by Howard and D'Angelo (1995); the system uses GP for identifying a model structure and GA for optimizing constants values.

Selection of Design Alternatives for Postfix-GP

We have used the postfix-based (linear) individual representation approach to improve the efficiency of Postfix-GP. We have implemented the following three different mechanisms in the Postfix-GP framework in order to improve population diversity and to avoid premature convergence: (i) avoid generation of semantically same individuals (duplicates) in the initial population, (ii) prevent mating between semantically same individuals, and (iii) use of semantic-aware subtree crossover operator. We have also used an archive to store the best solutions found so far. In the current implementation of Postfix-GP, we put a limit on the size of an individual in order to control the evolution of bloated solutions. Our future plan aims to include parsimony pressure and multiobjective selection mechanism in Postfix-GP for

controlling evolution of bloated solutions. Postfix-GP provides flexibility to the user to choose any of these two approaches for constant creation.

POSTFIX GENETIC PROGRAMMING FRAMEWORK

There are a number of GP tools such as GeneXproTools (Ferreira 2006), Open BEAGLE (Gagné and Parizeau 2002), ECJ (Luke et al. 2007), GPLab (Silva and Almeida 2003), and lil-GP (Zongker and Punch 1998) developed by GP practitioners for different platforms. Despite the number of tools available, none of them address the requirements of ease of use and small learning curve before utilizing them to solve the problems. Many of these tools are open source—Open BEAGLE (Gagné and Parizeau 2002); ECJ (Luke et al. 2007); GPLab for MATLAB (Silva and Almeida 2003); lil-GP (Zongker and Punch 1998)—and the rest are available commercially (GeneXproTools; Ferreira 2006). These tools are either able to solve a certain type of problem or demand actual source code alteration in order to create the required environment. Retrieval of the final solution, generated by these tools, requires translation of the output or digging of log files. Moreover, the final solution produced by these tools is difficult to interpret. Because of these reasons, we get motivated to develop our own GP framework (Dabhi and Vij 2011) that uses postfix notation for an individual representation.

Traditional GP represents individuals using a variable length nonlinear tree structure. The nonlinear tree structure is capable of representing a solution as an arithmetic expression or code written in programming language, depending on the type of problem to be solved. However, tree structure with pointer-based representation is not efficient because (i) recursive evaluation of a tree is time consuming, and (ii) pointer-based representation is not memory efficient. GP individual representation without any pointer reference reduces the required evaluation (fitness calculation) time. Moreover, this mechanism also minimizes the requirement of memory to store an individual in primary memory (Tokui and Iha 1999). To overcome the drawback of pointer-based representation and to improve performance of GP, linear node representation has been used by GP researchers. For example, Keith and Martin (1994) implemented a GP system in C++ using linear (prefix) representation. A variant of GEP, Prefix-GEP (Li et al. 2005) has employed prefix (fixed linear string) notation for an individual representation in order to preserve subcomponents of the fittest individuals.

We adopt postfix notation for an individual representation for our Postfix-GP framework. The genotype–phenotype mapping mechanism describes the relationship between genetic information of an individual and its behavior. We have used the mapping mechanism that obeys the rules of postfix notation expression. As discussed in Li et al. (2005), the use of Polish notation for an individual representation ensures that the nodes from

the same subtree of an expression tree appear adjoined to each other on a corresponding character string. Thus, there is a tight association between substructure pieces of genotype to subtrees of phenotype. This association is helpful to avoid destruction of good substructures of the fittest individuals during the evolution process.

Because the framework represents an individual in postfix notation, a stack can be used for evaluating the fitness of an individual. Use of a stack eliminates the need of conversion of an individual to a tree representation and then traversal of the tree for fitness evaluation. The algorithm that generates valid postfix individuals uses the following idea: when a random operand from a given terminal set is chosen, push it on the stack and add the chosen operand to the array list. When a random function (unary or binary) is chosen from a given function set, the number of arguments (operands) corresponding to the chosen function are popped from the stack and the chosen function is added to the array list. Implementation details of this algorithm are presented in Dabhi and Vij (2011).

Individual Structure and Length

To control growth in size of a solution (an individual), traditional GP imposes size or depth limits on the generated offspring. Approximating the size of a solution a priori (before GP starts its execution) is an open issue in the GP community. Poli, et al. (2008) suggested a two-step process to resolve this issue: (i) approximate the size of the minimal possible solution achievable by utilizing the given terminal and function set; (ii) add a safety margin of 50% to 100% to the size of the solution obtained in the previous step to estimate the final size of the solution. Determining the optimum size of the head of a gene is an open issue in original GEP. Still, there is no procedure available for finding the gene head size a priori for the given problem. The user has to run the GEP algorithm a number of times with different gene head sizes to find an acceptable solution.

To solve the issue of approximating solution size, we propose a fixed-length solution (individual) representation approach. However, the valid portion of an individual (forming a valid postfix equation) is determined dynamically using a validation algorithm. The valid portions of different individuals have different lengths. Individuals with different valid lengths produce postfix equations of different sizes and shapes. Individuals generated at the initialization step or offspring produced by the genetic operators are subject to the validation algorithm to determine their valid portion. Every individual in the Postfix-GP framework (Dabhi and Vij 2011) has three length attributes: MaxLength, MinLength, and ValidLength. The MinLength and MaxLength are user-defined attributes that define the range of lengths of syntactically valid individuals.

MinLength: Least length of a syntactically correct individual yielding valid postfix equation.
MaxLength: Maximum length of a syntactically correct individual yielding valid postfix equation.
ValidLength: Index (position) of the last element of an individual yielding valid postfix equation.

To discover the ValidLength of an individual, we have employed the concept of StackCount, presented by Keith and Martin (1994) and used by Tokui and Iha (1999) for implementation of linear genetic programming. The StackCount value of an element (node) of an individual is equal to the number of arguments it pushes onto the stack minus the number of arguments it pops off the stack. Thus, the StackCount value is equal to 1 for an operand, 0 for an unary operator, and −1 for a binary operator. The sum of StackCount values must be 1 at the ValidLength (index of the last element forming a valid postfix equation) position of an individual. Too-large size individuals take a long time to evaluate and consume precious computational resources. To prevent this, it is required to put a limit on the size of an individual. As Postfix-GP represents an individual using linear string in postfix notation, it is preferable to control the size of an individual by limiting its maximum length instead of its depth. Moreover, to compute the depth of an individual represented in linear form, conversion from a linear to a tree representation is required. This conversion increases overhead. Thus, to control growth in the size of an individual, we have set a limit on the maximum length of an individual.

The MaxLengths of individuals are fixed, but the ValidLengths of these individuals vary, producing individuals of different lengths and shapes. The ValidLength attribute of an individual refers to the index of the last element of an individual forming a valid postfix equation. For an individual to be considered Valid, its ValidLength may be equal to or less than the MaxLength of an individual, but must be greater than or equal to MinLength of an individual. Thus, an individual is referred as invalid if its ValidLength is less than MinLength or more than MaxLength. Figure 2 shows an individual in postfix notation, the syntactically valid portion of the individual and its tree representation. Postfix-GP (Dabhi and Vij 2011) does not require transformation of a syntactically valid portion of an individual to a tree form, nor traversal of the constructed tree, to calculate fitness of an individual. The tree representation is depicted only for better comprehension to the reader. As shown in Figure 2, ValidLength (syntactically valid termination point) of an individual is greater than MinLength, but less than MaxLength of an individual.

In spite of having fixed length, each individual has the potential to form postfix equations of different sizes and shapes. For example, if the user specifies that MinLength and MaxLength values are 1 and 12, then the smallest

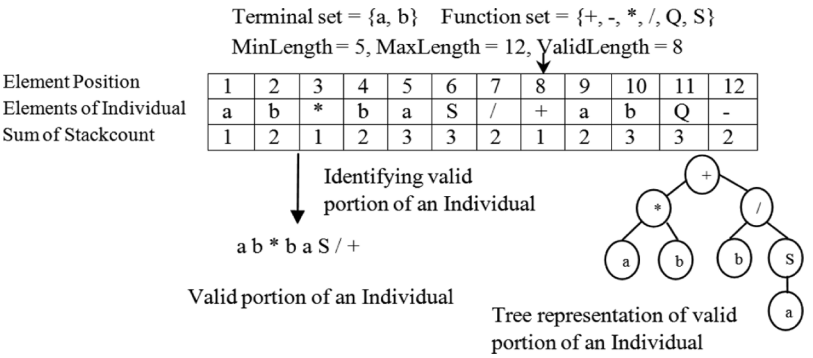

FIGURE 2 Postfix-GP individual representation.

individual (Figure 3), consisting of only one element and the largest consisting of as many elements as the MaxLength of an individual (Figure 4: when all the elements of an individual are forming a valid postfix equation), can be generated by Postfix-GP (Dabhi and Vij 2011). However, as shown in Figure 2, it is possible that an individual might have some elements that

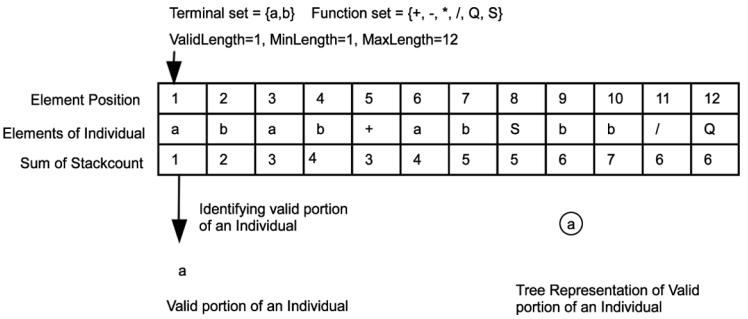

FIGURE 3 Smallest Postfix-GP individual with ValidLength equal to 1.

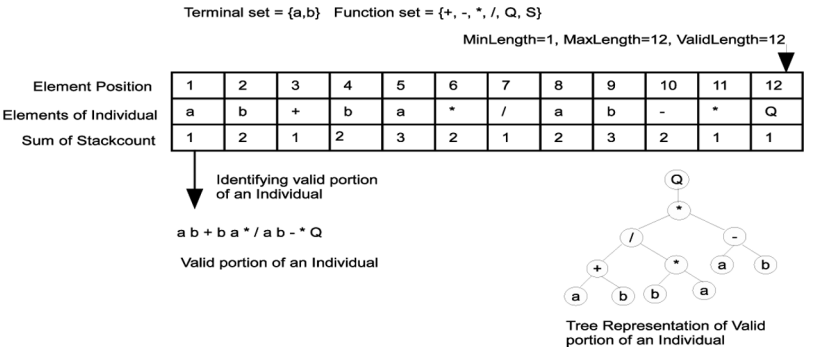

FIGURE 4 Largest Postfix-GP individual with ValidLength equal to 12.

are not useful for forming a syntactically valid postfix equation. These unused elements do not contribute to the fitness of an individual.

Initial Population

Postfix-GP uses a random initialization approach to generate an initial population of individuals having ValidLength between MinLength and MaxLength. Original GEP (Ferreira 2001) generates the initial population at random. However, Postfix-GP forces phenotypic diversity among the individuals of the initial population to better explore the search space in successive generations. Algorithm 1 presents the details for generating a phenotypic diverse initial population.

Algorithm 1 Phenotypic Diverse Initial Population

Input: MinLength, MaxLength, PopulationSize

$Count = 0$;
$CurrentPopulation \leftarrow \{\}$

while $Count < PopulationSize$ **do**
  $Individual\ \ temp \leftarrow GenerateIndividual(MinLength, MaxLength)$;
  $temp.AdjustedFitness \leftarrow CalculateFitness(temp)$;

  for $i = 1$ to $CurrentPopulation.Size$ **do**
    $Individual\ \ current = CurrentPopulation[i]$;
    if $((|temp.AdjustedFitness| - |current.AdjustedFitness|) == 0)$ **then**
      $temp \leftarrow GenerateIndividual(MinLength, MaxLength)$;
      $i \leftarrow 1$;
    end if
  end for

  $CurrentPopulation.Add(temp)$;
  $Count \leftarrow Count + 1$;
end while

Output: CurrentPopulation (ArrayList) consisting phenotypic diverse individuals

Genetic Operators

A common problem with GP is the generation of syntactically invalid solutions by genetic operators (Torres, Larre, and Torres 2002). Because of this problem, the solution search space remains unexplored and the performance of GP is affected (Russell et al. 1995; Torres, Larre, and Torres 2002). When designing a new genetic operator for GP, the following issues must be taken care of: (i) the operator should generate offspring with minimal bloat, and (ii) the operator should search solutions from a range of different sizes (lengths). The first requirement demands a fixed length solution (individual) representation

scheme; the second demands use of a variable length solution representation scheme. To satisfy these requirements and to explore a range of possible solutions (individuals of different lengths), Postfix-GP uses MinLength, MaxLength, and ValidLength attributes of an individual. In the next subsection, we present subtree and semantic-aware subtree crossover operators relevant to Postfix-GP.

Subtree crossover

Standard subtree crossover operator in GP has the following properties: (i) local search—offspring produced by the operator often inherit most of their genetic material from one parent; (ii) biased search—chosen crossover points are very close to the leaves of the tree. In our implementation of subtree crossover, crossover points for both parents are chosen randomly with values between 1 and ValidLength of parent. The subtrees (subexpressions) rooted at the chosen crossover points are extracted and swapped with each other. The following rule is applied for extracting the subtree from the parent: total sum of Stackcount values must be 1 at the end of a subtree. The steps for calculating the end position of a subtree are detailed in Dabhi and Chaudhary (2014). These steps are presented in Algorithm 2 for reference of the reader. The algorithm takes two arguments, an individual (parent) and a crossover point, and returns the end-position of the subtree, rooted at the supplied crossover point. If the lengths of extracted subtrees are different, then lengths of offspring will be different from the lengths of both parents. As depicted in Figure 5, the subtree crossover operator moves genetic material (in the form of a subtree) from one parent to a different position in the child. Thus, the crossover operator can alter position of the genetic material in the genotype.

Algorithm 2 Calculate End-Position of Subtree

Input: Parent p, SubTreeStartPoint cpt

$StackCountSum \leftarrow 0$
repeat
  if $(p[cpt]\ is\ operand\ ||\ p[cpt]\ is\ constant)$ **then**
    $StackCountSum \leftarrow StackCountSum + 1;$
  else if $(p[cpt]\ is\ BinaryOperator)$ **then**
    $StackCountSum \leftarrow StackCountSum - 1;$
  else if $(p[cpt]\ is\ UnaryOperator)$ **then**
    $StackCountSum \leftarrow StackCountSum;$
  end if
  $cpt \leftarrow cpt - 1;$
until $(StackCountSum! = 1)$

Output: SubTreeEndPoint cpt

Semantic-aware subtree crossover

It is highly important to produce solutions that are not only syntactically valid but are semantically valid during the evolutionary process of GP. Recently,

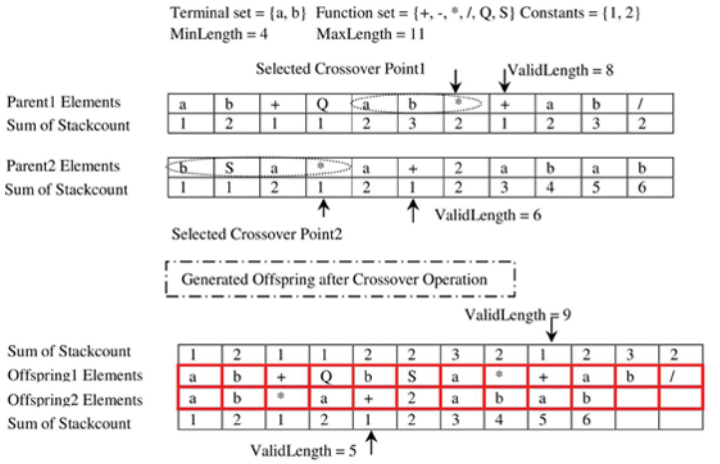

FIGURE 5 Subtree crossover in Postfix-GP.

semantics information of solutions has been used by practitioners to improve the searching ability of GP (Uy, Hoai, and O'Neill 2009). The semantic of a real-valued mathematical expression is the result it generates for a set of input values. The semantic of a solution describes its behavior with respect to a set of input values (Uy, Hoai, and O'Neill 2009). Two mathematical expressions that are the same in syntax must have the same semantics, but the reverse is not true. For example, look at the following four expressions:

$$z = x + x \quad (a) \quad \text{and} \quad z = x + x \quad (b) \tag{1}$$

$$z = x + x + x + x \quad (a) \quad \text{and} \quad z = 4 * x \quad (b) \tag{2}$$

Expressions (1a) and (1b) have the same syntax and so the same semantics. The expressions (2a) and (2b) are different in syntax, but they generate the same result and, hence, have the same semantics.

Several GP practitioners (Uy, Hoai, and O'Neill 2009; Uy et al. 2011) argue that semantic (behavioral) similarity rather than structural similarity of two subtrees to be swapped should be used while implementing crossover operator. The presumption behind the argument is that swapping of subtrees is most beneficial if these subtrees are not semantically similar. The semantic similarity of two subtrees is evaluated by comparing them on a set of given points in the solution space. We have used the notations Sampling Semantic (SS), Sampling Semantic Distance (SSD), and Semantic Equivalence (SE), described in Uy, Hoai, and O'Neill (2009) Uy et al. (2011), for implementing semantic-aware subtree crossover operator for Postfix-GP.

Uy, Hoai, and O'Neill (2009) and Uy et al. (2011) proposed a measure to compute semantic distance between two subtrees. They have defined

sampling semantic of a subtree as follows: let F as a function be expressed by a subtree ST on a domain D. Presume Q corresponds to a set of points from domain D, $Q = q_1, q_2, q_3, ., q_N$. Then SS (Uy, Hoai, and O'Neill 2009; Uy et al. 2011) of subtree ST on set Q of domain D is the set $R = r_1, r_2, \ldots, r_N$, where $r_i = F(q_i)$, $i = 1, 2, \ldots, N$. The points of set Q are used for calculating the sampling semantic of a subtree, and these points are sampled at random from the given set of fitness cases. The estimated semantic value will be more accurate, if the size of the set Q is large. However, because of the large set size, the semantic evaluation process will take more time. If the set size is too small, then the estimated semantic value will not be accurate. The size of the set Q is problem dependent. In all our experiments, at every generation, points of set Q are randomly selected from the given set of fitness cases to calculate the sampling semantic distance of two subtrees.

We follow Uy, Hoai, and O'Neill (2009) and Uy et al. (2011) to compute SSD between two subtrees by taking the sum of absolute differences for all values of SS and dividing the obtained sum by the number of SS points N. Let $X = \{x_1, x_2, \ldots, x_N\}$ and $Y = \{y_1, y_2, \ldots, y_N\}$ be the SS of subtree1 (subtr_1) and subtree2 (subtr_2) on the same set of sample points. SSD between subtr_1 and subtr_2 is:

$$\text{SSD}(\text{subtr}_1, \text{subtr}_2) = \frac{(|x_1 - y_1| + \cdots + |x_N - y_N|)}{N}$$

The value of SSD depends on the number of points N as well as on the scheme used for selecting these points from the given fitness cases. SSD is useful to check SE of two subtrees (Uy, Hoai, and O'Neill 2009; Uy et al. 2011). Two subtrees (subtr_1, subtr_2) are semantically equivalent if their sampling semantic distance is less than a defined threshold value (ϵ), named semantic sensitivity. The Semantic-Aware Crossover (SAC) operator checks the semantic equivalence of two subtrees before performing the crossover operation to promote semantic diversity of the population (Uy, Hoai, and O'Neill 2009).

We have used our earlier proposed semantic-aware subtree crossover operator (Dabhi and Chaudhary 2013), which prevents swapping of two subtrees having the same fitness values to improve search efficiency of the Postfix-GP framework. The operator ensures that the parents selected for crossover operation must be semantically different. The operator randomly selects a crossover point with value between 1 and ValidLength of parent. After selecting a crossover point for each parent, the next step is to extract the subtrees rooted at the selected crossover points in order to swap them. The rule that is applied to extract the subtree from the parent in the subtree crossover is used for this scheme as well. Semantic equivalence of extracted subtrees is checked using a set of sample data points. If the two subtrees are semantically equivalent, then new crossover points are chosen and the process is repeated until the subtrees are semantically different or the maximum number of trials have completed. An algorithm and

implementation details of semantic-aware subtree crossover operator for Postfix-GP is presented in Dabhi and Chaudhary (2013). Here, we made minor correction to the algorithm. The earlier version of the algorithm (Dabhi and Chaudhary 2013) swaps two subtrees before performing validation on the ValidLength of generated offspring. The modified algorithm validates the ValidLength of the offspring first, and if the ValidLength of both offspring respect the MinLength and the MaxLength constraints, then only it allows swapping of two subtrees. Algorithm 3 presents the modified semantic-aware subtree crossover algorithm.

Algorithm 3 Semantic Aware Subtree Crossover

Input: Parent p_1, Parent p_2

BEGIN
$Count \leftarrow 0$
repeat
$subtree_1.startpos \leftarrow$ *choose a random crossover point for* p_1
$subtree_2.startpos \leftarrow$ *choose a random crossover point for* p_2
$subtree_1.endpos \leftarrow EndPosition(p_1, subtree_1.startpos)$
$subtree_2.endpos \leftarrow EndPosition(p_2, subtree_2.startpos)$

$subtree_1.length \leftarrow subtree_1.startpos - subtree_1.endpos$
$subtree_2.length \leftarrow subtree_2.startpos - subtree_2.endpos$
$subtree_1 \leftarrow substring(subtree_1.endpos, subtree_1.length) forParent\ p_1$
$subtree_2 \leftarrow substring(subtree_2.endpos, subtree_2.length) forParent\ p_2$

Measure SSD between $subtree_1$ *and* $subtree_2$ *on setP*
$Count \leftarrow Count + 1$

if $(SE(subtree_1, subtree_2))$ **then**
continue;
else
$offspring_1.ValidLength \leftarrow p_1.ValidLength - subtree_1.length + subtree_2.length$
$offspring_2.ValidLength \leftarrow p_2.ValidLength - subtree_2.length + subtree_1.length$
end if

until $((offspring_1.ValidLength < MinLength || offspring_2.ValidLength < MinLength || offspring_1.ValidLength > MaxLength || offspring_2.ValidLength > MaxLength)\ \&\ \&\ (Count < MaxTrial))$

Swap $subtree_1$ *and* $subtree_2$
END

The performance comparison of three crossover operators (GA-like one point, subtree, semantic-aware subtree), on four symbolic regression problems, for Postfix-GP is presented in Dabhi and Chaudhary (2014). It is concluded that the semantic-aware subtree crossover significantly outperforms the other two crossover operators. The value of semantic sensitivity between 0.01 and 0.1 gives good results for semantic-aware subtree

crossover. We have used the modified semantic-aware subtree crossover algorithm (Algorithm 3) for all our experiments. We have set the value of semantic sensitivity to 0.1, for all experiments.

Mutation

Standard subtree mutation randomly selects a mutation point in a parent tree and swaps the subtree rooted at the selected point with a newly generated subtree. Another commonly used mutation is a point (node replacement) mutation, which imitates the single bit-flip mutation used in a genetic algorithm. Point mutation selects a mutation point randomly for an individual and replaces the selected element with a different element of the same arity. Postfix-GP framework uses point mutation. We have applied the point mutation to functional as well as terminal (variables and constants) elements. Current implementation of Postfix-GP uses the protected version of two functions (division by zero and square root of a negative number) to avoid run-time exceptions.

Selection mechanism

Postfix-GP framework has two selection mechanism: (i) roulette wheel selection and (ii) tournament selection. The roulette wheel selection mechanism is used for all experiments. In roulette wheel selection, all solutions are sorted and ranked according to their fitness values (goodness of fit). The roulette wheel selection strategy (Goldberg 1989) is used to pick out a subset of solutions (from the current population and the archive) that will undergo the action of genetic operators for creating a new generation of individuals (new population). Tournament selection runs a tournament among individuals randomly selected from the population. The winner of the tournament is used to produce offspring for the next generation. The size of the tournament decides the selection probability of an individual.

Gustafson et al. (2005) noted that the probability of "No change to fitness" events increases significantly when the selected parents for crossover have the same fitness values. Crossover of two dissimilar parents is likely to produce a change in offspring (solution) quality. To prevent mating between two solutions with the same fitness values, we have to select parents that have different behaviors (fitness values). Algorithm 4 presents the pseudocode for selecting behaviorally different parents. Postfix-GP uses an archive (Laumanns et al. 2002) to store "best-so-far" solutions. Crossover operation is performed between members selected from an archive and the current population.

Algorithm 4 Selection of Phenotypically Different Parents for Crossover

Input: Archive Population A, Current Population C, int $MaxTrial$

```
BEGIN
P1 ← Select an individual from Archive Population A
P2 ← Select an individual from Current Population C
Count ← 0
while (Count < MaxTrial) do
  if ((|P1.AdjustedFitness| − |P2.AdjustedFitness|) == 0) then
    P2 ← Select new individual from Current Population C
    Count ← Count + 1
  else
    break;
  end if
end while
END
```

Output: Phenotypically Different Parents

Fitness function

In the current implementation, all records of the training dataset are used to evaluate the fitness of every individual (solution). We have used mean absolute error (MAE) of an individual as a standardized fitness measure. MAE is defined as a measure of deviation of the predicted output from the observed one. MAE is not range bound and it cannot be negative. We have used adjusted fitness to measure the quality of an individual. The following equation is used for calculating the adjusted fitness:

$$\text{Adjusted Fitness} = 1/(1 + \text{Standardized Fitness}) \tag{3}$$

Adjusted fitness is range bound and has values in the range of 0 to 1, with a perfect fit corresponding to the value 1. The adjusted fitness is very useful to discriminate between individuals having close standardized fitness values, which may take place in generations near the end of the GP run.

Constants

To model an acceptable solution of a problem, the function and terminal sets must have all elements required to represent the solution (Koza 1992; Lopes and Weinert 2004). However, many times a modeler does not have full insight into the problem. Because of this, it is not possible to determine members of function and terminal sets in advance. This problem also occurs when selecting a set of constants, part of terminal set (Lopes and Weinert 2004). A study on GEP (Ferreira 2003) suggests two approaches for the constants creation. In the first approach, constants are created by the GEP algorithm itself. A second approach uses an explicit list of constants, as a part of terminal set,

TABLE 1 Difference Between Individual Representation Schemes of Postfix-GP and Other Evolutionary Approaches

EA	Genotye	Phenotype	Mapping
GA	Linear string of fixed length	Decoding genotype	Problem dependent
GP	Parse Tree	Parse Tree	Traversing Parse Tree
GEP	Linear string of fixed length	Parse tree	Karva notation
P-GEP	Linear string of fixed length	Expression tree	Prefix notation
EGIPSYS	Linear string with variable length	Expression tree	Karva notation
Postfix-GP	Linear string with variable length	Linear string	Postfix notation

which would be provided by a modeler. Postfix-GP framework provides the flexibility to the user to use either of these two approaches to create constants.

MODEL COMPLEXITY

To determine the size of an individual (solution) represented in a tree form, varieties of measures are defined by the GP community (Vladislavleva, Smits, and Hertog 2009): (i) the number of nodes in a tree, (ii) the number of layers in a tree, (iii) path length. We have used the number of nodes (elements) in a valid portion of a mathematical expression as a complexity measure. To measure the complexity of a model using this measure, we have to calculate the number of nodes (total number of variables, binary and unary functions) within a valid portion of a mathematical expression. In the current implementation of Postfix-GP, the ValidLength attribute represents this number and can be used directly to measure the complexity of an evolved model.

Table 1 presents the difference between an individual representation scheme of Postfix-GP approach and other evolutionary approaches.

EXPERIMENTS AND RESULTS

This section presents results of solving different empirical modeling problems using symbolic regression via Postfix-GP framework. Postfix-GP framework is developed using Microsoft .NET framework[1] on Windows XP operating system and all experiments reported in this article were run on a PC with AMD Athlon-XP 2.4 MHz processor and 2 GB of main memory. We have used ZedGraph[2] class library for plotting the charts. ZedGraph is an open source graph library for the .NET platform.

Postfix-GP is tested on two function identification and two deterministic chaotic time series modeling problems. We have presented the best result (solution) obtained out of 20 GP runs for all the problems. The obtained

[1]Microsoft .net framework software development kit, http://msdn.microsoft.com
[2]http://www.sourceforge.net/projects/zedgraph (2008)

solutions in infix notation along with their statistical measures are shown for each problem. We set the standard values for Postfix-GP parameters so that it can provide good performance on different problems. Solving complex problems using Postfix-GP may require setting of specific values for these parameters. Table 2 presents the GP parameter settings (Population Size = 500, No. of Generations = 100, Crossover Type = Semantic-aware subtree with semantic sensitivity = 0.1, MinLength = 10, MaxLength = 40).

We have used the correlation coefficient (Equation 6) to measure the closeness between the given set of points and those generated by the evolved solution (equation). The value of the correlation coefficient ranges from −1 to +1. The closer the value of r is to zero means there is less correlation between predicted and observed values. For each problem, we have also shown the evolved solution with its MAE (Equation 4), normalized mean squared error (Equation 5), adjusted fitness (Equation 3), structural complexity (size) of solution and correlation coefficient. The following equations are used to calculate these statistical measures, where $y_i, \hat{y}_i, Y, \hat{Y}, cov(Y, \hat{Y})$ and σ_Y^2 stand for the given ith observation, corresponding ith value calculated by the evolved model, actual observed series, estimated series, covariance between observed and estimated series, and variance of actual observed series.

$$\text{MAE} = \frac{1}{N}\left[\sum_{i=1}^{N}|(y_i - \hat{y}_i)|\right] \tag{4}$$

$$\text{NMSE} = \frac{1}{\sigma_Y^2}\left[\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2\right] \tag{5}$$

TABLE 2 Postfix-GP and lil-GP Parameters Setting

Parameter		Value	
		Postfix-GP	lil-GP
MaxLength	Problems 1 and 3	40	40
	Problem 2	50	50
	Problem 4	60	60
Population Size		500	5000
Crossover		Semantic-aware subtree	Subtree
Initial Population		Semantically diverse	half and half
Crossover rate		0.9	0.9
Mutation rate		0.1	0.1
Number of Generations		100	100
Function Set		{ +, −, *, / }	{ +, −, *, / }
Initial Tree Depth		–	2..6
keeps_trying		–	yes

$$r = \frac{cov(Y, \hat{Y})}{\sigma_Y \sigma_{\hat{Y}}}, \tag{6}$$

$$\text{where } \sigma_Y = \sqrt{\frac{\left(\sum_{i=1}^{N} (y_i)^2\right)}{N} - \left[\frac{\sum_{i=1}^{N} (y_i)}{N}\right]^2} \tag{7}$$

$$\text{Complexity (Size)} = \text{Valid Length of Solution}$$

Generally, machine learning practitioners use the performance of the selected model on the out-of-sample dataset as a measure to validate the selection. The out-of-sample (test) dataset used for validating the performance of a model should be different from that used for training (evaluating) a model. We used this approach for validating the selection of a model. We validated an evolved model by measuring its goodness-of-fit on in-sample (training) and out-of-sample (unseen) datasets. For each problem, the evolved model was used to make an out-of-sample prediction. We measured the performance of an evolved model on unseen dataset by calculating statistical measures such as MAE, NMSE, and correlation coefficient.

Sequence Induction Series

Problem 1: Explicit use of constant list

We have taken the problem of the Sequence Induction Series with constants. The problem is solved using the GEP approach in Ferreira (2003). Equation (8) is used to generate the induction series

$$b = 5a^4 + 4a^3 + 3a^2 + 2a + 1, \tag{8}$$

where a consists of nonnegative integers. For this problem, the first ten positive integers (values of independent variable a) in the range [1,10] and their corresponding values (values of dependent variable b), obtained using Equation (8), are used as fitness cases (training dataset).

$$\begin{aligned} b = &((((((a - 0.2) * a) + (0.8 + a))/0.2) + -1) * (a * a)) \\ &+ (-0.7 + (a + ((a + (((a - 0.3) * a) + a))/a)))) \end{aligned} \tag{9}$$

$$b = 5a^4 + 4a^3 + 3a^2 + 2a + 1 \tag{10}$$

For this experiment, terminal set T consists of an independent variable and a list of constants: T = {a, list of constants}. Equation (9) represents one of the best solutions found by Postfix-GP. Simplifying this equation results in Equation (10), which is similar to Equation (8).

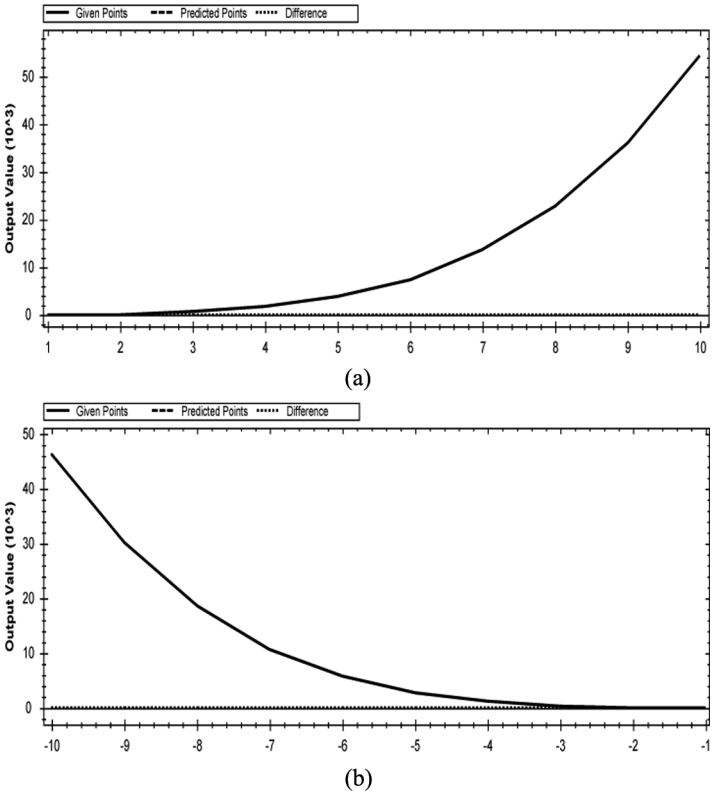

FIGURE 6 Postfix-GP with explicit list of constants for the induction series: comparison of actual series with the series modeled by the evolved solution for (a) 10 training data points (b) one-step-ahead predictions for 10 data points.

Figure 6(a) depicts the comparison of the actual function with the function modeled by the evolved solution. The evolved solution has structural complexity (number of nodes) of 33 and MAE = 0.0006, NMSE = 5.96×10^{-15}, AdjustedFitness = 0.9993, and $r = 0.9999$ for the training dataset. The X-axis of the Figure 6(a) represents the point number, not the real value of the point. Figure 6(b) shows out-of-sample (test data) one-step-ahead predictions using the evolved solution for the values of a in the range [$-10{:}1{:}-1$]. We found MAE = 0.0003, NMSE = 2.06×10^{-15}, AdjustedFitness = 0.9996, and $r = 0.9999$ for the test dataset. The errors are nearer to zero value, suggesting that the one-step-ahead predictions are very good.

Problem 2: Creating constant from scratch

A second experiment is performed on the induction series problem to evolve the solution without using an explicit list of constants. The terminal set T = $\{a\}$ is used for this experiment. Here, we are interested in checking the ability of the Postfix-GP system to generate constants from scratch.

TABLE 3 Induction Series: Statistical Measures for the Best Evolved Postfix-GP Solutions Obtained Using Two Different Constant Creation Approaches

	Training			One-Step-Ahead		
Approach	MAE	NMSE	r	MAE	NMSE	r
With Constants	0.0006	5.96×10^{-15}	0.9999	0.0003	2.06×10^{-15}	0.9999
Without Constants	0	0	1	0	0	1

The best solution found by Postfix-GP has 47 nodes. The solution is presented in Equation (11). Simplifying this equation results in Equation (12), which is an almost complete match for the target, Equation (8). We got MAE = 0 and NMSE = 0 for the training dataset. The MAE = 0, NMSE = 0, AdjustedFitness = 1 and $r = 1$ for one-step-ahead predictions of 10 data points (for the values of a in the range [−10:1:−1]). Table 3 shows values of MAE, NMSE and r for the training dataset and one-step-ahead predictions of 10 data points obtained using evolved solutions for both constant creation approaches.

$$b=(((((a+(((a+a)+a)-a))+((a+a)-(a/a)))*(((((a/a)+a)+a)-a)+(a*a)))*a)+((a+(a/a))+(a-((a*a)-a)))) \tag{11}$$

$$b = 5a^4 + 4a^3 + 3a^2 + 2a + 1 \tag{12}$$

Noisy Quadratic Function

Problem 3

We have taken the problem of Noisy Quadratic Function, which is presented and discussed in (Lopes and Weinert 2004). The noisy quadratic function problem is a synthetic problem of a polynomial regression. For this problem, noise is generated synthetically and added to the real output, to make the output corrupted. As discussed in Lopes and Weinert (2004), a total of 201 data points are generated by using Equation (13).

$$y = 2x^2 - 3x + 4 + \text{Noise}, \tag{13}$$

where Noise is generated using following equation:

$$\text{Noise} = \left(\frac{\text{random}}{5}\right) - 0.1,$$

where random is any random number generated in the range [0, 1]. The values for independent variable, $x(i)$, are obtained using an equation $x(i+100) = \sin(i/10)$, with $i = -100, \ldots, 100$.

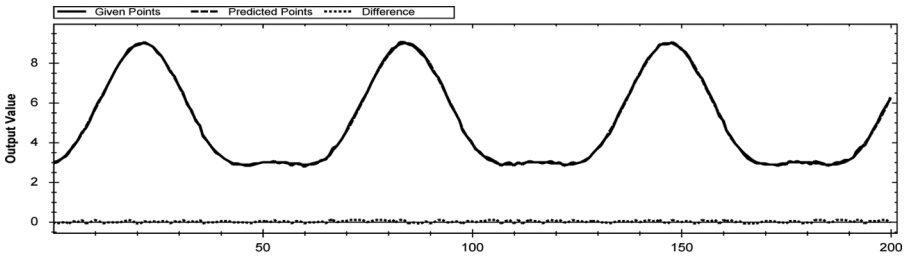

FIGURE 7 Noisy Quadratic Series: Comparison of 200 training (actual) data points with the points modeled by the evolved solution.

For this problem, we have used an explicit list of constants in the range [−1, 1]. The terminal set T = {x, list of constants}. Equation (14) represents one of the best solutions found by Postfix-GP. Simplifying this equation results in Equation (15), which is very close to the Equation (13). Figure 7 compares the actual function with the function modeled by the evolved solution. The evolved solution has structural complexity of 29. We obtained MAE = 0.0481, AdjustedFitness = 0.9540, and r = 0.999 with the training

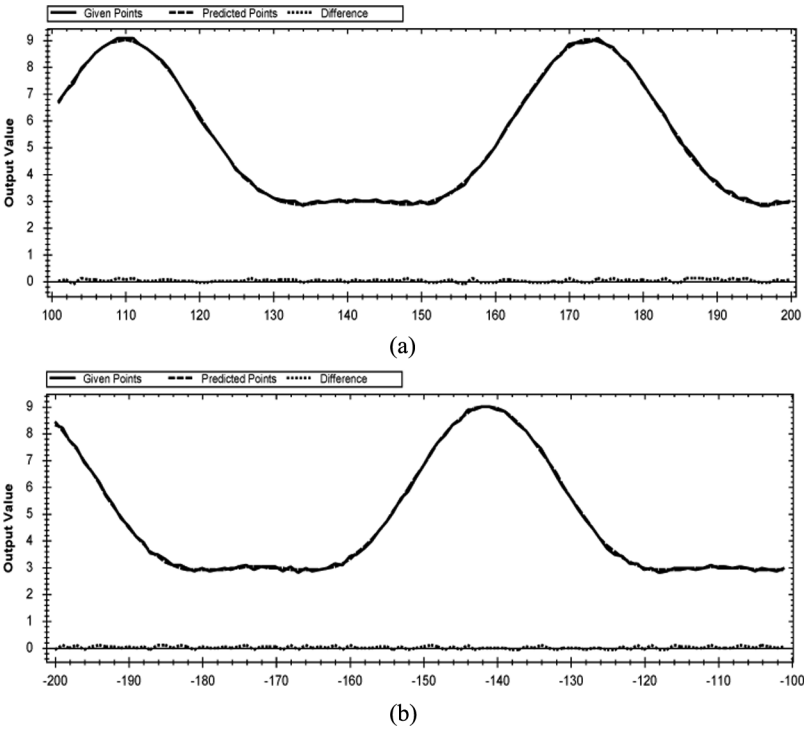

FIGURE 8 Noisy Quadratic Series: Comparison of 100 actual data points with one-step-ahead predictions for values of (a) i in range [100:1:200]; (b) i in range [−200:1: −100].

TABLE 4 Noisy Quadratic Series: Statistical Measures for One-Step-Ahead Predictions Obtained Using the Best Evolved Postfix-GP Solution

Data Points Starting	Ending	One-Step-Ahead Predictions MAE	NMSE	r
101	200	0.0476	0.0005	0.9997
−201	−100	0.0502	0.0007	0.9996

dataset. Figure 8(a) shows out-of-sample (test data) one-step-ahead prediction made using the evolved solution on 100 data points for the values of i in the range [100:1:200]. Figure 8(b) shows out-of-sample (test data) one-step-ahead predictions made using the evolved solution on 100 data points (for values of i in the range [−200:1:−100]). Table 4 shows the values of MAE, NMSE, and r for one-step-ahead predictions done using the evolved solution for the values of i in the range [100:1:200] and [−200:1: −100].

$$y = ((((((-1 + x) + x) * (x * -0.6))/ - 0.6) - ((-0.7 + (x + 0.4)) + ((-0.6/(-0.6/ - 0.2)) + x))) + (-0.7/ - 0.2)) \tag{14}$$

$$y = 2x^2 - 3x + 4 \tag{15}$$

Nonlinear Chaotic Series

Problem 4

A system is referred to as a dynamical system if its present state is somehow dependent on its previous states. A nonlinear dynamical system is a system in which the dependence of the present state on previous states can be represented by a nonlinear combination of previous states. Constructing a model of a nonlinear dynamical system from the observed time series is a challenging problem. A chaotic system is a nonlinear dynamical system with the property of sensitive dependence on initial conditions.

Both logistic map and Rossler system are examples of deterministic chaotic systems. Modeling of these nonlinear chaotic systems from their external behavior is a challenging task. Because of the chaotic nature of these systems, the evolved GP model gives good performance for short-term prediction of future values. We have used the notation x_i to represent x_{t-i} term. The final result is obtained by algebraic simplification of an evolved Postfix-GP model.

Logistic map

Logistic map is a one-dimensional map used to illustrate chaos. For a logistic map, the value of the next data point x_{j+1} is dependent on the current data

point x_j and the control parameter. The logistic map is defined by

$$X_{n+1} = rX_n * (1 - X_n), \tag{16}$$

where r is a control parameter. The problem is referred to and solved using GP in Ahalpara and Parikh (2006). We set the value of r to 3.891 and X_0 to 0.1 to generate the chaotic time series. We have bypassed the initial 2000 transient points, as done in Ahalpara and Parikh (2006), and taken the next 500 data points (with value of $d = 1$ and $\tau = 1$) as the training dataset. For this problem, we have used an explicit list of constants in the range $[-1, 1]$ with a precision of one digit. For this experiment, terminal set $T = \{X_n$, list of constants $\}$ is used.

$$\begin{aligned} X_{n+1} = (((((0.1 * (((0.1 * ((0.7 + 0.8) * (0.1 + (0.1 + 0.1))))) \\ - 0.7) - 0.8)) - 0.8) - (-1/-1)) * (X_n + X_n)) * (X_n + -1)) \end{aligned} \tag{17}$$

$$X_{n+1} = 3.891X_n * (1 - X_n) \tag{18}$$

The best solution found by Postfix-GP had 31 nodes and is presented in Equation (17). Simplifying this equation results in Equation (18), which is similar

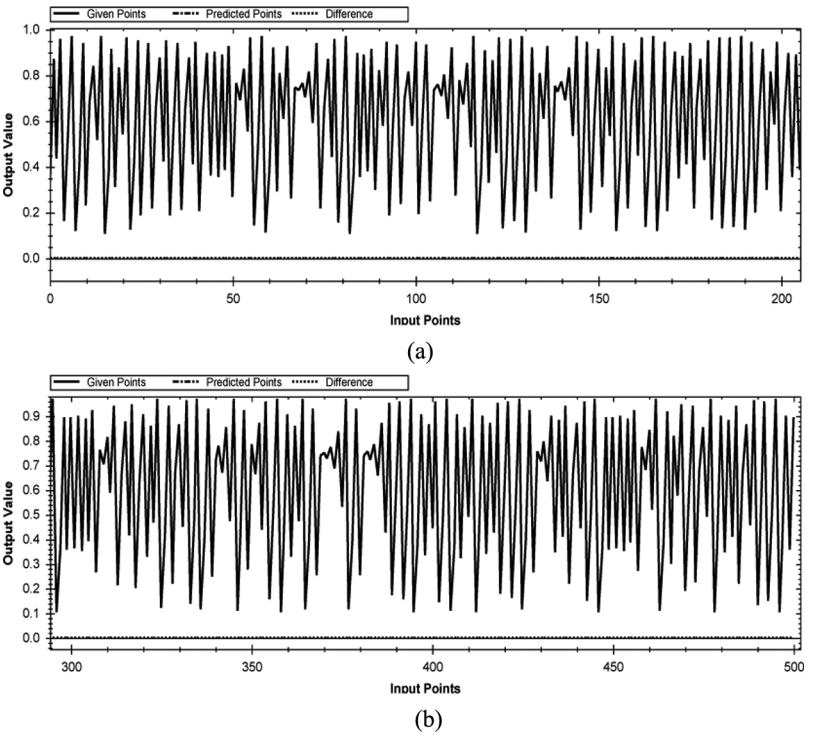

FIGURE 9 Logistic map: Comparison of actual series with the series modeled by the evolved solution for training data points. (a) First 200 points, (b) last 200 points.

to the Equation (16), used for generating the training dataset. Figure 9 presents comparison of training data points with the points modeled by the evolved Postfix-GP. We got MAE = $3.43^{*}10^{-8}$, NMSE = $2.53^{*}10^{-14}$, Adjusted Fitness = 0.9999, and $r = 0.9999$ for the training dataset.

The evolved model is used to perform out-of-sample one-step-ahead, and multistep-ahead (dynamic) predictions of unseen data points. Figure 10(a) shows out-of-sample one-step-ahead predictions made using the evolved model on the dataset of 200 points, starting at data point 6000. The MAE and NMSE values for these 200 data points are 3.41×10^{-8} and 2.56×10^{-14}. The model's out-of-sample one-step-ahead predictions are very close to the given data points. Figure 10(b) shows out-of-sample dynamic predictions for 35 points starting at data point 500. It is observed that dynamic predictions are very good up to 25th point. After that point, prediction accuracy goes down. The reason for this decline in prediction accuracy is the positive Lyapunov exponent of the series. For the logistic series, the

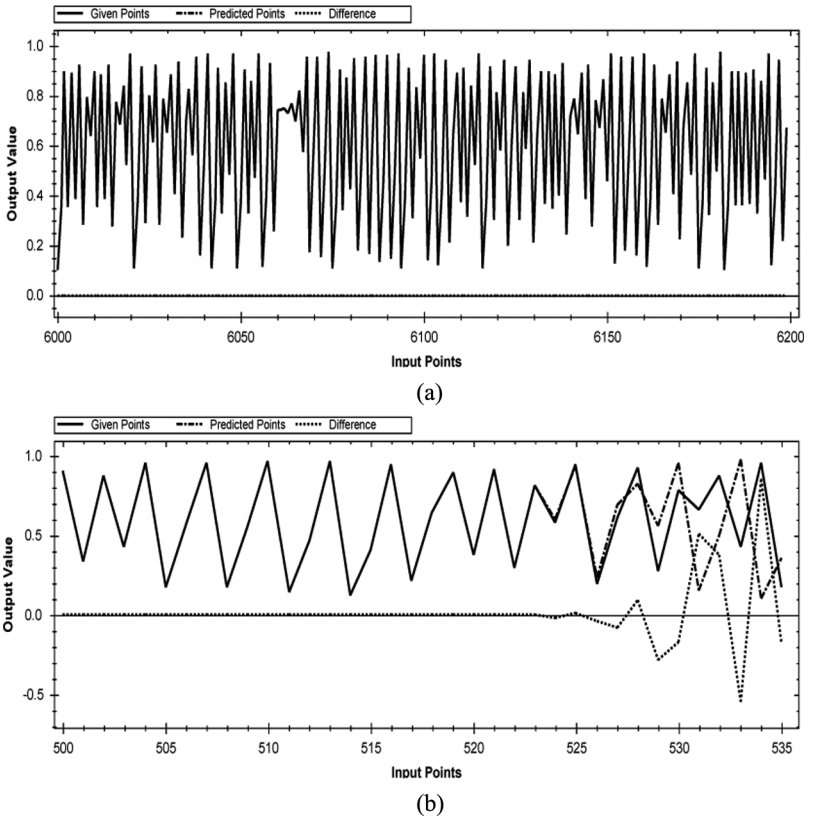

FIGURE 10 Comparison of actual data points with out-of-sample predictions for logistic series: (a) One-step-ahead predictions of 200 data points (b) Multistep-ahead predictions of 35 data points.

value of maximum Lyapunov exponent is 0.48. When the time series has a positive Lyapunov exponent, there is a time horizon beyond which the prediction goes down (Ahalpara and Parikh 2006). Using the mentioned value of the Lyapunov exponent, an initial error of 5.0×10^{-7} in the first step of multistep predictions grows to 0.2 in 27th steps. The same behavior is observed in multistep predictions made using the evolved model.

Rossler system

The Rossler system is an example of a chaotic attractor in three-dimensional space. The Rossler system has three-parameters: a, b, and c. When these parameters are assigned to specific values, the series demonstrates, a chaotic property. The following three equations are used to generate the series (Ahalpara and Parikh 2006).

$$\begin{aligned} X_{t+\Delta} &= X_t - [Y_t + Z_t]\Delta. \\ Y_{t+\Delta} &= Y_t + [X_t + aY_t]\Delta. \\ Z_{t+\Delta} &= Z_t + [b + X_t Z_t - cZ_t]\Delta. \end{aligned} \tag{19}$$

$$\begin{aligned} x_t = (((2-(1-(((x_2-((x_3-(x_4/-3))-(((-9-((((((-6-(((x_2-((x_5-x_1) \\ +-3))/-7)*(x_2-x_1)))/-7)*(x_2-x_1))^{-}-7)-x_2))/(-7-1)) \\ *(-9-x_1))))/-9)*(x_2-x_1))))/-9)-(x_2-x_1)) \end{aligned} \tag{20}$$

The time series of X_t, Y_t, and Z_t are generated by setting initial values of $x_0 = -1$, $y_0 = 0$, $z_0 = 0$, $\Delta = 0.02$, $a = 0.2$, $b = 0.2$, and $c = 5.7$ (Ahalpara and Parikh 2006). We have used the series of X variable and taken values of every 50th point to generate a new series for X, as described in (Ahalpara and Parikh 2006; Szpiro 1997). We have reconstructed a series from the generated series of X with values of time delay $\tau = 1$ and embedding dimension $d = 8$ (Szpiro 1997). We used 500 data points as the training dataset. The best solution found by Postfix-GP had 59 nodes and is presented in Equation (20). We got MAE = 0.6147, NMSE = 0.0641, AdjustedFitness = 0.6192 and $r = 0.9697$ for the training dataset. Simplification of this equation results in Equation (21).

$$x_t = \frac{-1}{9} - \frac{1}{81}\left[\left(x_3 + \frac{x_4}{3} + \frac{(x_p+9)(x_1+9)}{8} - x_2\right) * (x_2 - x_1)\right] - (x_2 - x_1), \tag{21}$$

where $x_p = \left[\frac{6}{7} - \frac{(x_2 - x_5 + x_1 + 3) * (x_2 - x_1)}{49}\right] * (x_2 - x_1) + 7 - x_2$.

Figure 11(a) shows out-of-sample one-step-ahead predictions made using the evolved Postfix-GP model on 100 data points (501 to 600). We

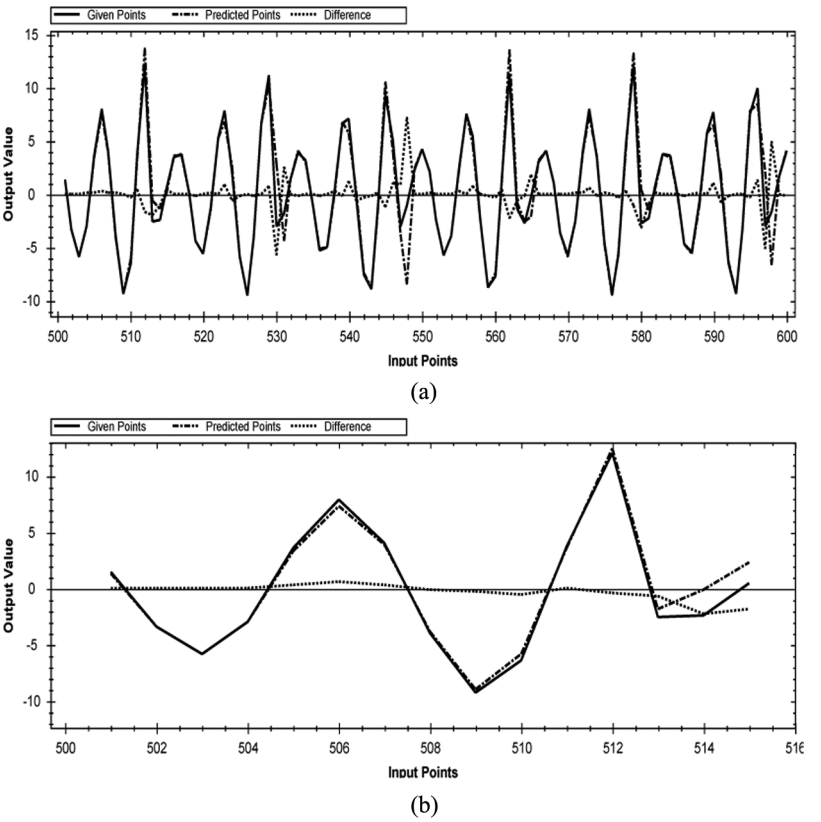

FIGURE 11 Comparison of actual data points with out-of-sample predictions for Rossler X-Series: (a) One-step-ahead predictions of 100 data points (b) Multi-step-ahead predictions of 15 data points.

obtained MAE = 0.6116 and NMSE = 0.0648 for one-step-ahead predictions of these 100 data points. The model's out-of-sample one-step-ahead predictions are very close to the given data points.

Figure 11(b) shows out-of-sample dynamic predictions for 15 points starting at point 500. We obtained MAE = 0.5013 and NMSE = 0.0206 for dynamic predictions of 15 data points. It is observed that dynamic predictions are very good up to the 13th point. After that point, predictions deteriorate. The reason for this drop in prediction accuracy is the positive Lyapunov exponent of the series. The value of largest Lyapunov exponent for the considered series is 0.2. With this value of Lyapunov exponent, an initial error of 0.09 in the first step of multistep predictions grows to 1.80 in 15th steps. The same behavior is observed in multistep predictions made using the evolved model.

Table 5 presents the MAE, NMSE, and correlation coefficient values for one-step-ahead prediction and dynamic prediction for the logistic map and the Rossler system.

TABLE 5 Out-of-sample Prediction Performance of Evolved Postfix-GP Solutions for Logistic and Rossler Series: (a) One-Step-Ahead Predictions (b) Dynamic Predictions

	One-Step-Ahead			Dynamic		
Series	MAE	NMSE	r	MAE	NMSE	r
Logistic	3.41×10^{-8}	2.56×10^{-14}	0.9999	0.0880	0.4767	0.7616
Rossler	0.6116	0.0648	0.9681	0.5013	0.0206	0.9905

COMPARISON OF RESULTS OBTAINED BY POSTFIX-GP AND LIL-GP

We have compared prediction performance of the Postfix-GP system with a popular GP system, lil-GP (Zongker and Punch 1998), lil-GP is implemented using C programming language. The lil-GP system does not provide a graphical user interface to load the training data. The system uses a parameter file (input.file) to load the GP parameters. The lil-GP generates six reporting files (.bst, .stt, .gen, .prg, .his, .sys), which provide statistical information of the GP run (Zongker and Punch 1998). Moreover, the system also generates one (.fn) file for symbolic regression problems. We have used lil-GP version 1.1, which is freely available, for all the experiments. We have made changes to application specific source files (function.h, function.c, app.c) of lil-GP to extend the function and terminal sets. By default, lil-GP uses the sum of absolute errors between the observed and the approximated data points as the standard fitness measure. The definition of fitness measure is modified to take the mean of the sum of absolute error as a new fitness measure.

We have not considered constants as a part of the terminal set for the induction series problem and let both systems evolve the constants. We have taken the values of constants in the range of -1 to $+1$ and as a part of the terminal set for the noisy quadratic and logistic map problems. The range of constants for the Rossler system is -10 to $+10$.

We followed the following two steps, suggested by Poli et al. (2008), for estimating the size of a solution: (i) find out the minimum possible solution, achievable using the given terminal and function sets; (ii) add a safety margin of 50%–200% to the size of the solution obtained in the previous step. Following this suggestion, we set the maximum size (maximum number of nodes parameter of lil-GP and MaxLength of PostfixGP) to 40 for noisy quadratic and logistic problems, whereas it was set to value 50 and 60 for the induction and the Rossler series problems.

The method for generation of the initial population is set to half and half for lil-GP. The half of the population is generated using grow and the other half using full approach. The minimum and maximum tree depth values, for generation of the initial population, are set to 2 and 6. We kept the "keeps_trying" parameter on for lil-GP. When this parameter is on, it

continues selection of new crossover points on the selected two parents until it produces child trees that do not violate the specified node and depth limits.

We have compared the performance of Postfix-GP with lil-GP on the previously discussed four problems. We have used the following measures for performance comparison: (i) correlation coefficient r, (ii) number of generations required by the system to evolve the best solution Gen_{best}, (iii) number of elements (nodes) of the best solution $\text{Node}_{\text{best}}$, and (iv) adjusted fitness of the best solution $\text{AdjFit}_{\text{best}}$. We have used $\text{Node}_{\text{best}}$ and Gen_{best} measures because these two measures are useful in the calculation of the average computational effort required by the GP to find the best solution of the run. We did 20 independent runs for each problem and presented the mean and the standard deviation values for these performance measures. Table 2 presents the parameters settings for Postfix-GP and lil-GP, used for all the experiments.

Table 6 presents results for the induction and noisy series. Both Postfix-GP and lil-GP found good solutions for the induction series problem. The average value of correlation coefficient for both the systems is very close to the value 1 for this problem. However, lil-GP requires a higher number of generations (Gen_{best}) to find the best solution compared to Postfix-GP. The sizes of the best solutions found by both the systems are similar. The mean value of adjusted fitness is higher for Postfix-GP compared to lil-GP.

As shown in Table 6, both Postfix-GP and lil-GP found good solutions for the noisy quadratic function problem. The average value of correlation coefficient for both the systems is equal to the value 1. Both systems require almost the same number of generations to find the best solution. The sizes of the best solutions found by each systems are almost the same.

Table 7 presents results for the logistic and Rossler series. Postfix-GP produced good results compared to lil-GP for the logistic map problem. The average value of best adjusted fitness for Postfix-GP is high (0.995) compared to the value for lil-GP (0.948). A very small value of standard deviation for mean adjusted fitness indicates that Postfix-GP is able to find better solutions consistently. Moreover, the average value of NMSE for Postfix-GP is small compared to lil-GP.

According to Table 7, both lil-GP and the Postfix-GP fail to evolve good solutions for the Rossler series. Very high value of average NMSE and low

TABLE 6 Results of 20 Runs for the Best Evolved Postfix-GP and lil-GP Solutions for Training Dataset: (i) Induction and (ii) Noisy

Problem	System	NMSE	$\text{AdjFit}_{\text{best}}$	Gen_{best}	$\text{Node}_{\text{best}}$	r
Induction	lil-GP	0.0 ± 0.000	0.409 ± 0.363	52.75 ± 33.927	47.1 ± 3.463	0.999 ± 0.000
	Postfix-GP	0.0 ± 0.000	0.703 ± 0.381	36.9 ± 30.839	47.9 ± 2.198	0.999 ± 0.000
Noisy	lil-GP	0.001 ± 0.00	0.953 ± 0.002	71.05 ± 23.141	30.3 ± 6.334	1.0 ± 0.00
	Postfix-GP	0.001 ± 0.00	0.952 ± 0.007	78.85 ± 15.699	38.3 ± 2.273	1.0 ± 0.00

TABLE 7 Results of 20 Runs for the Best Evolved Postfix-GP and lil-GP Solutions for Training Dataset: (i) Logistic and (ii) Rossler

Problem	System	NMSE	$\text{AdjFit}_{\text{best}}$	Gen_{best}	$\text{Node}_{\text{best}}$	r
Logistic	lil-GP	0.064 ± 0.045	0.948 ± 0.017	30.35 ± 29.571	17.7 ± 9.543	0.985 ± 0.019
	Postfix-GP	0.002 ± 0.005	0.995 ± 0.009	67.6 ± 21.886	36.7 ± 4.366	0.999 ± 0.002
Rossler	lil-GP	0.498 ± 0.068	0.38 ± 0.01	25.95 ± 23.476	7.9 ± 3.582	0.823 ± 0.008
	Postfix-GP	0.134 ± 0.063	0.50 ± 0.06	94.105 ± 5.666	58.7 ± 0.73	0.937 ± 0.064

value of mean best adjusted fitness indicate that lil-GP was not able to come out from the local minima. The mean NMSE value for the Postfix-GP system is much less compared to lil-GP. Moreover, the value of the mean correlation coefficient is high for Postfix-GP compared to lil-GP.

To measure and compare the predictive performance of evolved Postfix-GP and lil-GP models, we use a classical measure: best adjusted fitness; see Equation (3). Table 8 shows the mean and standard deviation values for the best adjusted fitness over 20 runs. We have tested the normality of the best adjusted fitness values, produced by both Postfix-GP and lil-GP models, for one-step-ahead predictions. We applied the Kolmogorov–Smirnov (K-S) test to check the normality of the best adjusted fitness values distributions. The K-S test gives very small p-values, which suggests that the values do not follow a normal distribution. Therefore, we applied a nonparametric test, the Wilcoxon test, with $\alpha = 0.05$ to evaluate the significance of obtained results. Table 8 presents Wilcoxon analysis of the obtained results. According to Table 8, no significant difference was observed for induction and noisy series. However, the mean best adjusted fitness values for Postfix-GP is higher than lil-GP for the induction series, whereas it is almost equal for the noisy series. The p-value for both of the chaotic series is lower than 0.05, indicating that the one-step-ahead predictive power of evolved Postfix-GP models is significantly better than that of lil-GP models.

We can conclude that the quality of solutions found by Postfix-GP was better than the quality of those found by lil-GP for all the problems. Moreover, the population size for Postfix-GP was set to 500, 10 times lower than that used for lil-GP (5000). Thus, the mean computational effort required by Postfix-GP was much lower than that of lil-GP. The mean best adjusted

TABLE 8 Wilcoxon Test for Comparing One-Step-Ahead Prediction Performance of Best Evolved lil-GP and Postfix-GP Solutions for Four Problems; Results Obtained Over 20 Independent Runs

Problem	lil-GP	Postfix-GP	Wilcoxon test	Significant?
Induction	0.3320 ± 0.3910	0.5688 ± 0.4635	p-value $= 0.3415$	No
Noisy	0.9532 ± 0.0025	0.9523 ± 0.0045	p-value $= 0.0948$	No
Logistic	0.9475 ± 0.0153	0.9955 ± 0.0085	p-value $= 8.4e-08$	Yes
Rossler	0.3709 ± 0.0049	0.5057 ± 0.0686	p-value $= 6.7e-08$	Yes

fitness values are higher for evolved Postfix-GP solutions compared to lil-GP solutions for all the problems.

CONCLUSION

We have thoroughly presented implementation details of Postfix-GP, including an individual representation, different crossover operators, and selection mechanism. Two approaches, one with explicit use of a list of constants and another without using the list, for the constant creation in Postfix-GP, are presented with examples. The Postfix-GP was applied to two problems of function identification and two problems of chaotic time series. We presented the Postfix-GP evolved models with their statistical details and models' out-of-sample (test data) prediction capability. The one-step prediction accuracy, obtained by evolved models, is very good for all the problems, which suggests the effectiveness of this method. The results are satisfactory for multistep predictions of chaotic time series.

The one-step-ahead prediction performances of Postfix-GP solutions were compared with those obtained using lil-GP. The significance of the results is tested by applying the Wilcoxon test on best adjusted fitness measure. The results suggest that one-step-ahead predictions performance of Postfix-GP solutions is better than lil-GP solutions for chaotic time series (logistic and Rossler), whereas no significant difference was observed for function identification (induction and noisy series) problems. However, the mean best adjusted fitness values are higher for evolved Postfix-GP solutions compared to lil-GP solutions, for all the problems. These results suggest that the Postfix-GP produced good quality solutions, consistently better than lil-GP for the tested problems.

REFERENCES

Ahalpara, D. P., and J. C. Parikh. 2006. Modeling time series of real systems using genetic programming. http://arxiv.org/PS_cache/nlin/pdf/0607/0607029v1.pdf.

Altwaijry, N., and M. El Bachir Menai. 2012. Data structures in multi-objective evolutionary algorithms. *Journal of Computer Science and Technology* 27:1197–1210.

Babovic, and M. Keijzer. 2000. Genetic programming as a model induction engine. *Journal of Hydroinformatics* 2:35–60.

Dabhi, V., and S. Chaudhary. 2013. Semantic subtree crossover operator for postfix genetic programming. In *Proceedings of Seventh International Conference on Bio-Inspired Computing: Theories and Applications (BIC-TA 2012), Advances in Intelligent Systems and Computing*, 201:391–402. India: Springer India.

Dabhi, V., and S. Chaudhary. 2015. Empirical modeling using genetic programming: A survey of issues and approaches. *Natural Computing* 14:303–330.

Dabhi, V., and S. Vij. 2011. Empirical modeling using symbolic regression via postfix genetic programming. In *Proceedings of the international conference on image information processing (ICIIP 2011)*, 1–6. IEEE.

Dabhi, V. K., and S. Chaudhary. 2014. Performance comparison of crossover operators for postfix genetic programming. *International Journal of Metaheuristics* 3:244–264.

Ferreira, C. 2001. Gene expression programming: a new adaptive algorithm for solving problems. *Complex Systems* 13:87–129.

Ferreira, C. 2002. Gene expression programming in problem solving. In *Soft Computing and Industry*, 635–653. London: Springer.

Ferreira, C. 2003. Function finding and the creation of numerical constants in gene expression programming. In *Advances in Soft Computing*, 257–265, Berlin: Springer.

Ferreira, C. 2006. *Gene expression programming: mathematical modeling by an artificial intelligence*. Vol. 21, 2nd ed. New York: Springer.

Gagné, C., and M. Parizeau. 2002. Open BEAGLE: A new C++ evolutionary computation framework. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, 888. New York, NY: Morgan Kaufmann.

Goldberg, D. E. 1989. *Genetic Algorithms in Search, Optimization and Machine Learning*. Boston, MA, USA: Addison-Wesley.

Gustafson, S., E. Burke, and N. Krasnogor. 2005. On improving genetic programming for symbolic regression. In *The 2005 IEEE congress on evolutionary computation*, 1:912–9. IEEE.

Handley, S. 1994. On the use of a directed acyclic graph to represent a population of computer programs. In *Evolutionary computation: Proceedings of the first IEEE conference on the IEEE world congress on computational intelligence*, 1:154–159. IEEE.

Holland, J. H. 1992. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. Cambridge, MA, USA: MIT Press.

Howard, L., and D. D'Angelo. 1995. The ga-p: A genetic algorithm and genetic programming hybrid. *IEEE Expert: Intelligent Systems and Their Applications* 10:11–15.

Keijzer, M. 2004. Alternatives in subtree caching for genetic programming. In *Genetic Programming*, 328–337. Springer.

Keith, M. J., and M. C. Martin. 1994. Genetic programming in C++ : Implementation issues. In *Advances in genetic programming*, ed. K. E. Kinnear Jr. Kinnear, K. E. Jr. (ed.), 285–310. Cambridge, MA, USA: MIT Press.

Koza, J. R. 1992. *Genetic Programming: On the programming of computers by means of natural selection*. Vol. 1. Cambridge, MA, USA: MIT Press.

Laumanns, M., L. Thiele, E. Zitzler, and K. Deb, 2002. Archiving with guaranteed convergence and diversity in multi-objective optimization. In *Proceedings of the genetic and evolutionary computation conference (GECCO '02)*, 439–447. San Francisco, CA, USA: Morgan Kaufmann.

Li, X., C. Zhou, W. Xiao, and P. C. Nelson. 2005. Prefix gene expression programming. In *Late breaking paper at genetic and evolutionary computation conference (GECCO '05)*, 25–31. Washington, D.C.: GECCO.

Lopes, H. S., and W. R. Weinert, 2004. EGIPSYS: An enhancgene expression programming approach for symbolic regression problems. *International Journal of Applied Mathematics and Computer Science* 14:375–384.

Luke, S., L. Panait, G. Balan, et al. 2007. ECJ 16: A java-based evolutionary computation research system. https://cs.gmu.edu/∼eclab/projects/ecj/

O'Neill, M., L. Vanneschi, S. Gustafson, and W. Banzhaf. 2010. Open issues in genetic programming. *Genetic Programming and Evolvable Machines* 11:339–363.

Poli, R., W. B. Langdon, and N. F. McPhee. *A field guide to genetic programming*. 2008. UK Ltd: Lulu.

Rosca, J. P. 1995. Entropy-driven adaptive representation. In *Proceedings of the workshop on genetic programming: From theory to real-world applications*, 23–32. San Francisco, CA: Morgan Kaufmann.

Russell, S. J., P. Norvig, J. F. Canny, J. M. Malik, and D. D. Edwards. *Artificial intelligence: a modern approach*. 1995. Vol. 74, Englewood Cliffs, NJ, USA: Prentice hall.

Ryan, C., and M. Keijzer. 2003. An analysis of diversity of constants of genetic programming. In *Proceedings of the 6th European conference on Genetic programming, EuroGP'03*, 404–13, Berlin, Heidelberg: Springer-Verlag.

Silva, S., and J. Almeida. 2003. Gplab-a genetic programming toolbox for matlab. In *Proceedings of the Nordic MATLAB conference*, 273–278.

Silva, S., and E. Costa. 2009. Dynamic limits for bloat control in genetic programming and a review of past and current bloat theories. *Genetic Programming and Evolvable Machines* 10:141–79.

Smits, G., and E. Vladislavleva. 2006. Ordinal Pareto genetic programming. In *IEEE congress on evolutionary computation (CEC 2006)*, 3114–20. IEEE.

Szpiro, G. G. 1997. Forecasting chaotic time series with genetic algorithms. *Physical Review E* 55:2557–68.

Tokui, N., and H. Iha. 1999. Empirical and statistical analysis of genetic programming with linear genome vol. 3, *Proceedings of IEEE international conference on systems, man, and cybernetics*, 610–615.

Torres, S., M. Larre, and J. Torres. 2002. A string representation methodology to generate syntactically valid genetic programs. In *WSEAS Transactions on Systems* 1:290–295.

Uy, N. Q., N. X. Hoai, and M. O'Neill. 2009. Semantic aware crossover for genetic programming: The case for real-valued function regression. In *Proceedings of the 12th European conference on genetic programming, EuroGP '09*, 292–302. Berlin, Heidelberg: Springer-Verlag.

Uy, N. Q., N. X. Hoai, M. O'Neill, R. I. Mckay, and E. Galván-López. 2011. Semantically-based crossover in genetic programming: Application to real-valued symbolic regression. *Genetic Programming and Evolvable Machines* 12:91–119.

Vladislavleva, E. J., G. F. Smits, and D. Den Hertog. 2009. Order of nonlinearity as a complexity measure for models generated by symbolic regression via Pareto genetic programming. *Transactions on Evolutionary Computing* 13:333–49.

Yao, X., and Y. Xu. 2006. Recent advances in evolutionary computation. *Journal of Computer Science and Technology* 21:1–18.

Zhang, B. T., and D. Y. Cho. 1998. Genetic programming with active data selection. In *Selected papers from the 2nd Asia-Pacific conference on simulated evolution and learning SEAL'98*, 146–153. London, UK: Springer-Verlag.

Zongker, D., and B. Punch. 1998. lilgp 1.1 genetic programming system. http://garage.cse.msu.edu/software/lil-gp/ (accessed January 13, 2014).